SPML Assignment 4 - Machine Learning

Clemens Beissel 4547330
Christian Lammers 4578236

August 26, 2018

1 Overall specification

For this machine learning exercise a n x m world with obstacles was used. The obstacles consisted of walls. Furthermore did this world consist of so called absorbing states, or states with a reward. The reward can be negative or positive. An example for such a world is the figure 1a.

Both algorithms that had to be programmed for this exercise can be applied to such a world. For the purpose of checking the work of the algorithm, a modification was made to the grid world. The utility values of the state are being displayed in their associated state. This makes checking the algorithm much easier.

2 Value iteration algorithm

2.1 Short description of the algorithm

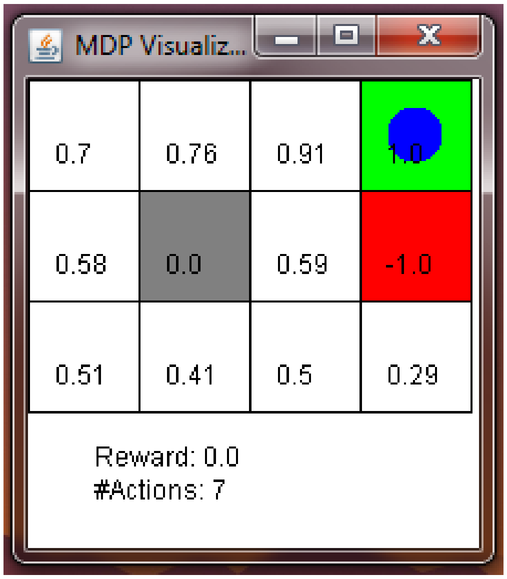
The value iteration algorithm recursively calculates the goodness or the utility of a state s. This calculation is based on the reward of entering that state s and the product of some discount factor γ together with the transition probability p that gives us the probability to enter the successor state s$'$ from state s with action a and the reward for entering the successor state s$'$.

If the world is non-deterministic performing one action does not always lead to the desired result. Therefore the algorithm has to consider all possible actions that can be applied in state s. For the purpose of calculating the utility of a state, the action with the highest value is chosen.

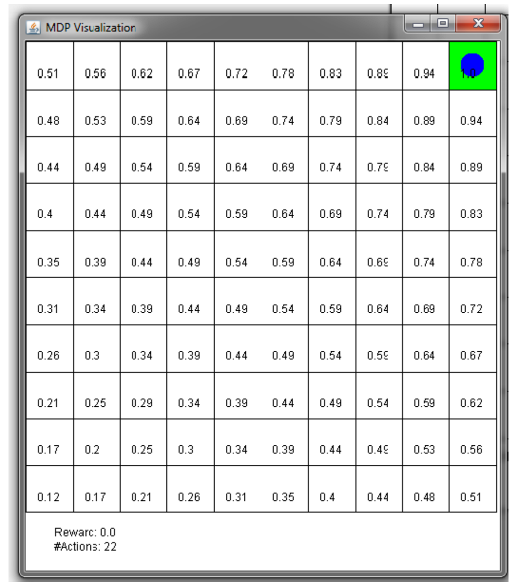
2.2 Experimenting with different worlds

For every world in this section the values of the states converge. The algorithm finds a policy that leads the agent to the desired goal state.

The agent follows the path with the greatest number, namely from state (1,1) straight to (1,3) and then right to the goal state (4,3) which can also be seen in figure 1a.

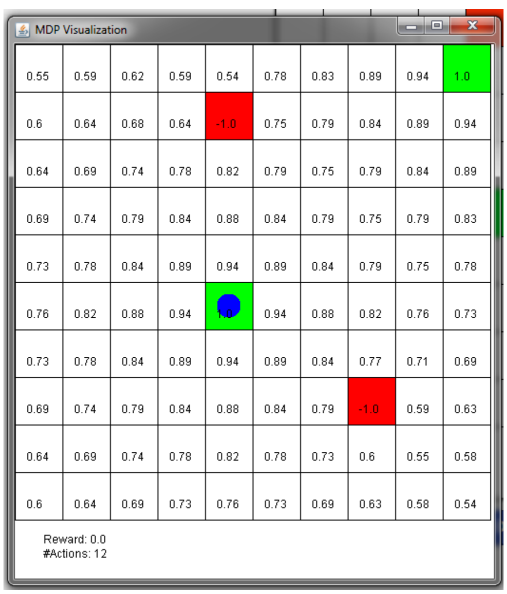


(a) 4x3 world

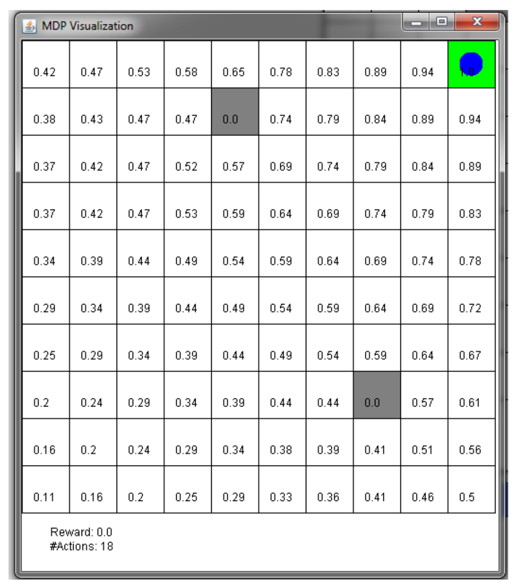


(b) 10x10 world with winning state at (10,10)

Figure 1: Two different grid worlds



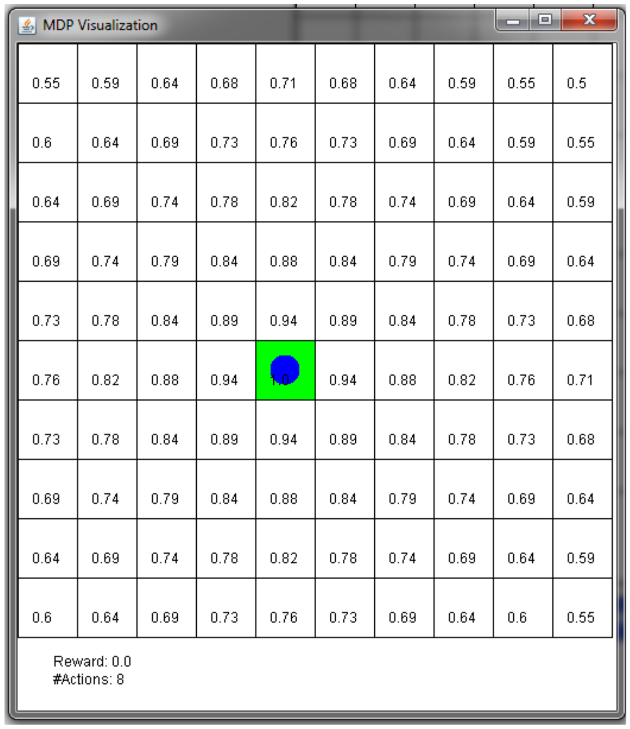
(a) 10x10 world with two negative and positive rewards



(b) 10x10 world with two obstacles

Figure 2: Two different grid worlds

Figure 3: 10x10 world with positive reward at (5,5)

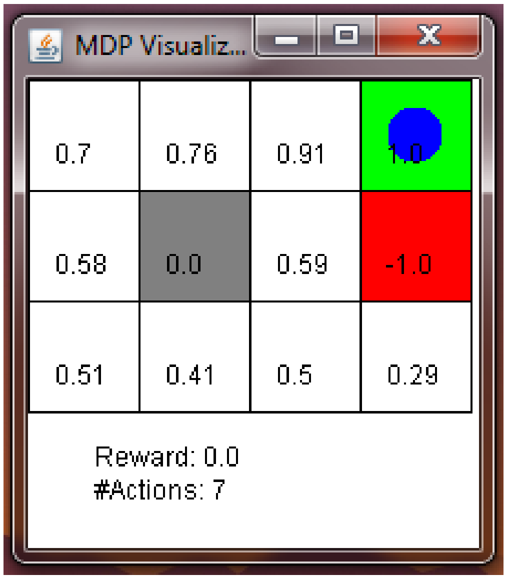


For the other worlds the agent also finds the optimal way to the goal state which is also reflected by the values printed on the state field.

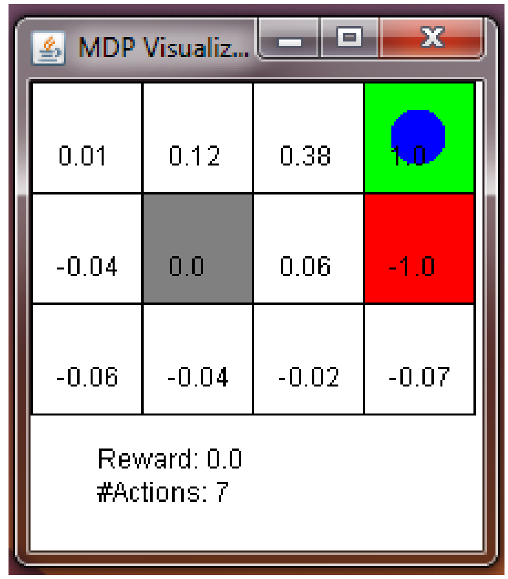
2.3 Experimenting with different values of the discount factor

Experimenting with different values of the discount factor on the 4x3 world with a threshold value of 0.0000001 lead to following results:

After the discount factor was lowered, the difference between the expected values is more subtle. With a discount factor of 0.1 almost all values are hardly any different from each other thus the agent has a hard time following the policy with success. The greater the discount factor is, the closer the expected values are correlated to the values of the absorbing states which can be seen in figure 4a and 4b.



(a) 4x3 world with discount factor 1.0

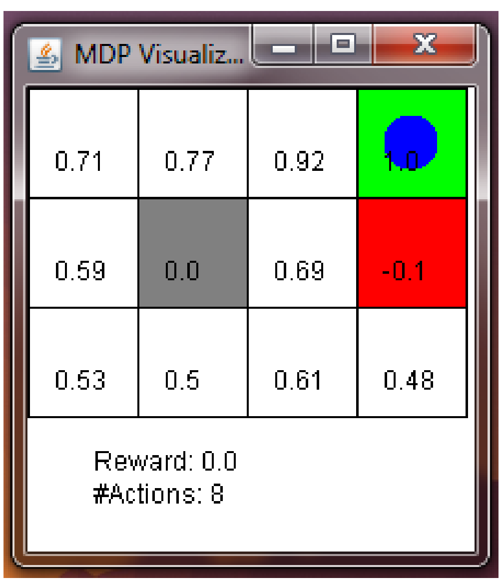


(b) 4x3 world with discount factor 0.5

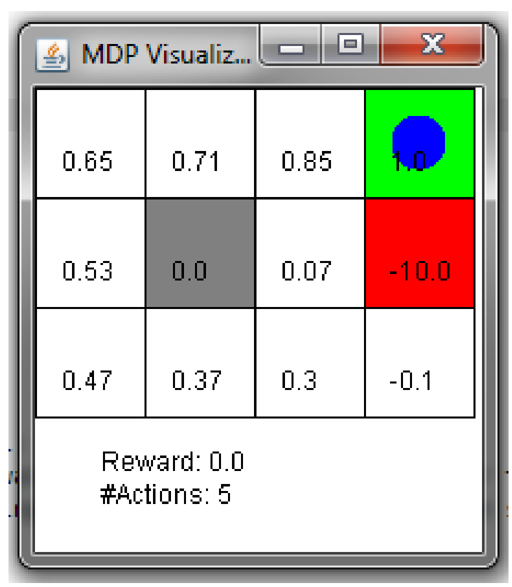
Figure 4: Two different grid worlds

2.4 Experimenting with different values for the state penalty

For the tests with different state penalty values the 4x3 grid world was used with a threshold value of 0.0000001:
As can be seen in figure 5b, do the overall expected values become more negative the lower the negative reward is set (as expected). However the agent does not have any difficulties finding a good path to the positive reward, no matter how negative the expected values become, because there is still the positive absorbing state which is influencing the surrounding states in a positive manner.



(a) 4x3 world with state penalty -0.1



(b) 4x3 world with state penalty -10

Figure 5: Two different grid worlds

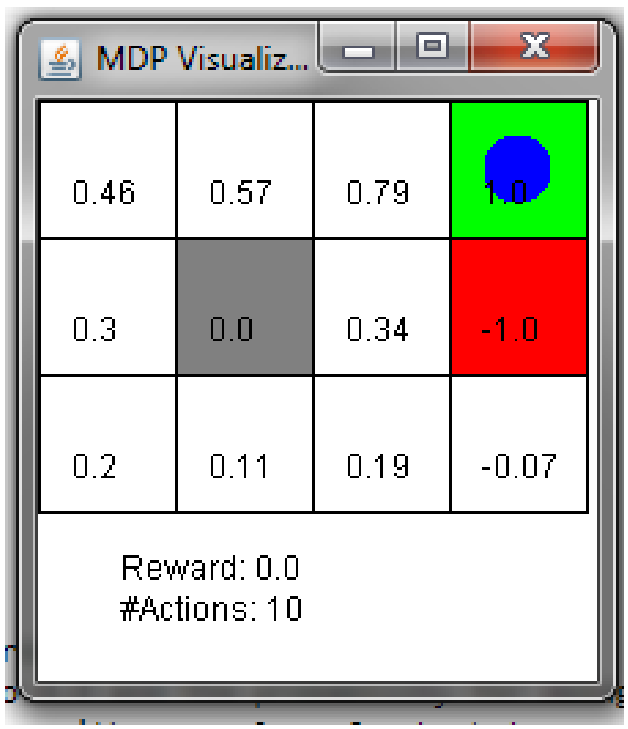
2.5 Experimenting with different settings of the transition probabilities

Normally the probability for moving in the desired direction is set to 0.8 and the probability for doing a sidestep is set to 0.2. Thus moving backwards and staying at the same state is never a possibility. After some experimenting with the transition probabilities new values where established for the backstep and the nostep probability:

pPerform = 0.5;
pSidestep = 0.2;
pBackstep = 0.1;
pNoStep = 0.2;

The same values as in the previous tests were used for all the other parameters. One can see in figure 6 that the expected values are slightly different from the ones with the original transition probabilities. The overall path, however, does not change due to the unchanged position of the absorbing states. Even after setting the backstep probability to 0.9 the path still does not change.

Figure 6: 4x3 world with different transition probabilities



3 Q-learning algorithm

3.1 Short description of the algorithm

In contrast to the value iteration algorithm, the Q-learning algorithm evaluates the quality of every possible action per state. The agent has to perform actions to receive the corresponding reward and learns through its history of state → action → reward experiences what actions to take in what state.
This idea was implemented with a three dimensional array where the first two dimensions refer to the location of the state and the third dimension refers to the action. There was a problem with the synchronization of the threads which lead to an incorrect execution of the program when normally running it. This was fixed by inserting a thread.sleep in the loop of the algorithm.

3.2 Comparison with exercise 1

Both of the algorithm found a path for the 4x3 grid world that the agent can use to reach the goal state. Both of these paths were found in reasonable time and no significance was found between the performances.

However, comparing the performance of the two algorithms in the 10x10 grid world with each other, shows some obvious differences. The value iteration algorithm converges much faster than the Q-learning algorithm which is mostly due to the actual carrying out of the actions during the Q-learning algorithm. This carrying out of the actions takes much more time and as a consequence does the algorithm need much more time to converge.

A Code used for both algorithms

```
package nl.ru.ai.vroon.mdp;

/**
 *
 * @author Clemens Beissel 4547330
 */
public class Tuple {

    private Action a;
    private Double value;

    public Tuple(Action a, Double v) {
        this.a = a;
        value = v;

    }

    public Action getAction() {
        return a;
    }

    public Double getValue() {
        return value;
    }

    public void setValue(Double v){
        value=v;
    }

}
```

```
package nl.ru.ai.vroon.mdp;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.math.BigDecimal;
import java.math.RoundingMode;

import javax.swing.JPanel;

/**
 * Creates visual content in accordance with the given MDP
 * @author Sjoerd Lagarde + some adaptations by Jered Vroon
 *
```

```
 */
public class DrawPanel extends JPanel {

	private static final long serialVersionUID = 1L;
	private int screenWidth;
	private int screenHeight;
	private MarkovDecisionProblem mdp;
	private ValueIterationAlgorithm va;
	private QLearningAlgorithm ql;
	private boolean valueIteration;

	/**
	 * Constructor
	 * @param mdp
	 * @param screenWidth
	 * @param screenHeight
	 */
	public DrawPanel(MarkovDecisionProblem mdp,ValueIterationAlgorithm va, int
	     screenWidth, int screenHeight) {
		this.mdp = mdp;
		this.va = va;
		this.screenWidth = screenWidth;
		this.screenHeight = screenHeight;
		this.valueIteration = true;

	}
	public DrawPanel(MarkovDecisionProblem mdp,QLearningAlgorithm ql, int
	    screenWidth, int screenHeight) {
		this.mdp = mdp;
		this.ql = ql;
		this.screenWidth = screenWidth;
		this.screenHeight = screenHeight;
		this.valueIteration = false;
	}


	@Override
	public void paintComponent(Graphics g) {
		setBackground(new Color(255, 255, 255));		// White
		    background
		super.paintComponent(g);

		int stepSizeX = screenWidth/mdp.getWidth();
		int stepSizeY = screenHeight/mdp.getHeight();

		Graphics2D g2 = (Graphics2D)g;
		for ( int i=0; i<mdp.getWidth(); i++ ) {
			for ( int j=0; j<mdp.getHeight(); j++ ) {
				Field f = mdp.getField(i, j);

				g2.setPaint(Color.WHITE);
				if ( f.equals(Field.REWARD) ) {
					g2.setPaint(Color.GREEN);
				} else if ( f.equals(Field.NEGREWARD) ) {
					g2.setPaint(Color.RED);
				} else if ( f.equals(Field.OBSTACLE) ) {
					g2.setPaint(Color.GRAY);
				}
				g2.fillRect(stepSizeX*i, screenHeight - stepSizeY
```

```
                                *(j+1), stepSizeX,stepSizeY);


                            if ( mdp.getStateXPosition() == i && mdp.getStateYPostion() == j ) {
                                g2.setPaint(Color.BLUE);
                                g2.fillOval(stepSizeX*i+stepSizeX/4, screenHeight - stepSizeY*(j+1)+stepSizeY/4, stepSizeX/2, stepSizeY/2);
                            }

                            g2.setPaint(Color.BLACK);
                            g2.drawRect(stepSizeX*i, screenHeight - stepSizeY*(j+1), stepSizeX,stepSizeY);
                            if(valueIteration)
                            g2.drawString(String.valueOf(round(va.getExpectedValue(i, j),2)), stepSizeX*i +12, screenHeight - stepSizeY*(j) -12);

                        }
                }
                g2.drawString("Reward: \t\t"+mdp.getReward(), 30, screenHeight+25);
                g2.drawString("#Actions: \t\t"+mdp.getActionsCounter(), 30, screenHeight+40);
        }

            private double round(double value, int places) {
            if (places < 0) throw new IllegalArgumentException();

            BigDecimal bd = new BigDecimal(value);
            bd = bd.setScale(places, RoundingMode.HALF_UP);
            return bd.doubleValue();
}

}
```

B Code for the value iteration algorithm

```
package nl.ru.ai.vroon.mdp;

import java.util.ArrayList;
import java.util.Collection;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.stream.Collectors;

/**
 *
 * @author christianlammers
 */
public class ValueIterationAlgorithm {

    MarkovDecisionProblem mdp;

    Double discount = 1.0;
```

```
    Double sigma = 0.0000001;
    int counter = 0;
    private int maxIterations = 10000;
        Double[][] stateUtilities;

    private Action[][] policy;

    public ValueIterationAlgorithm(MarkovDecisionProblem m) {
        mdp = m;
        mdp.setInitialState(0, 0);
        policy = new Action[mdp.getWidth()][mdp.getHeight()];
        stateUtilities = new Double[mdp.getWidth()][mdp.getHeight()];

    }

    /*
    The core value Iteration algorithm
     */
    public void valueIteration() {
        Map<Action, Double> utils;
        Double[][] oldStateUtilities = new Double[mdp.getWidth()][mdp.getHeight()];
        initStateUtility(oldStateUtilities);
        Boolean done = false;

        while (!done || counter >= maxIterations) {

            //loop through every state
            for (int x = 0; x < mdp.getWidth(); x++) {
                for (int y = 0; y < mdp.getHeight(); y++) {
                    double reward = mdp.getReward(x, y);
                    if (reward != mdp.getNoReward()) { //if the current state is a sink or an obstacle
                        stateUtilities[x][y] = reward;
                    } else {

                        utils = getUtils(x, y, oldStateUtilities); // returns map of actions and their values
                        Tuple bestAction = getBestAction(utils, oldStateUtilities[x][y]); // returns the best action and the best value based on utils
                        stateUtilities[x][y] = reward + bestAction.getValue() * discount; // gives the current state a new expected value (bellmann equation)

                        policy[x][y] = bestAction.getAction(); //adds the best action for every state to the list.
                    }

                }


            }
            done = checkDifference(stateUtilities, oldStateUtilities, sigma);
            oldStateUtilities = duplicate(stateUtilities);
            System.out.println(counter);
            counter++;
        }

```

```java
        for (int x = 0; x < mdp.getWidth(); x++) {
            for (int y = 0; y < mdp.getHeight(); y++) {
                System.out.println("(" + x + "," + y + ")" + " : " +
                    stateUtilities[x][y] + " Action: " + policy[x][y]);

            }
        }

    }

    /*
    returns a HashMap of Actions and their corresponding expected values
        originating from the position x y.
     */
    private Map getUtils(int x, int y, Double[][] oldStateUtilities) {
        Map<Action, Double> utils = new LinkedHashMap<>();
        Double thisState = oldStateUtilities[x][y];
        int right = x + 1;
        int left = x - 1;
        int up = y + 1;
        int down = y - 1;

        if (right < mdp.getWidth()) {
            utils.put(Action.RIGHT, oldStateUtilities[right][y]);
        } else {
            utils.put(Action.RIGHT, thisState);
        }

        if (left >= 0) {
            utils.put(Action.LEFT, oldStateUtilities[left][y]);
        } else {
            utils.put(Action.LEFT, thisState);
        }

        if (up < mdp.getHeight()) {
            utils.put(Action.UP, oldStateUtilities[x][up]);
        } else {
            utils.put(Action.UP, thisState);
        }

        if (down >= 0) {
            utils.put(Action.DOWN, oldStateUtilities[x][down]);
        } else {
            utils.put(Action.DOWN, thisState);
        }

        return utils;
    }

    /**
     * Sets all expected values to 0.0
     *
     * @param stateValues
     */
    private void initStateUtility(Double[][] oldStateUtilities) {

        for (int i = 0; i < mdp.getWidth(); i++) {
            for (int j = 0; j < mdp.getHeight(); j++) {
                oldStateUtilities[i][j] = 0.0;
```

```java
            }
        }
    }

    /*
    returns the best Action and the corresponding expected value for a state.

     */
    private Tuple getBestAction(Map<Action, Double> utils, Double valueOfThisState) {
        Double[] actionValues = new Double[4];

        actionValues[0] = utils.get(Action.UP) * mdp.getpPerform()
                + utils.get(Action.nextAction(Action.UP)) * mdp.getPSideStep() / 2
                + utils.get(Action.previousAction(Action.UP)) * mdp.getPSideStep() / 2
                + utils.get(Action.backAction(Action.UP)) * mdp.getpBackstep()
                + valueOfThisState * mdp.getPNoStep();
        actionValues[1] = utils.get(Action.RIGHT) * mdp.getpPerform()
                + utils.get(Action.nextAction(Action.RIGHT)) * mdp.getPSideStep() / 2
                + utils.get(Action.previousAction(Action.RIGHT)) * mdp.getPSideStep() / 2
                + utils.get(Action.backAction(Action.RIGHT)) * mdp.getpBackstep()
                + valueOfThisState * mdp.getPNoStep();
        actionValues[2] = utils.get(Action.DOWN) * mdp.getpPerform()
                + utils.get(Action.nextAction(Action.DOWN)) * mdp.getPSideStep() / 2
                + utils.get(Action.previousAction(Action.DOWN)) * mdp.getPSideStep() / 2
                + utils.get(Action.backAction(Action.DOWN)) * mdp.getpBackstep()
                + valueOfThisState * mdp.getPNoStep();
        actionValues[3] = utils.get(Action.LEFT) * mdp.getpPerform()
                + utils.get(Action.nextAction(Action.LEFT)) * mdp.getPSideStep() / 2
                + utils.get(Action.previousAction(Action.LEFT)) * mdp.getPSideStep() / 2
                + utils.get(Action.backAction(Action.LEFT)) * mdp.getpBackstep()
                + valueOfThisState * mdp.getPNoStep();

        Double bestValue = -999999999999.0;
        int bestIndex = 0;
        for (int i = 0; i < actionValues.length; i++) {
            if (actionValues[i] > bestValue) {
                bestValue = actionValues[i];
                bestIndex = i;
            }
        }

        Action bestAction = null;
        switch (bestIndex) {
            case 0:
                bestAction = Action.UP;
                break;
            case 1:
                bestAction = Action.RIGHT;
                break;
            case 2:
                bestAction = Action.DOWN;
```

```java
                break;
            case 3:
                bestAction = Action.LEFT;
        }

        return new Tuple(bestAction, bestValue);
    }

    /*
    returns true if the difference between the current expected value and the
        previous expected value is less than sigma for all states.
     */
    private Boolean checkDifference(Double[][] stateUtilities, Double[][]
        oldStateUtilities, Double sigma) {
        Double difference = 0.0;
        for (int x = 0; x < mdp.getWidth(); x++) {
            for (int y = 0; y < mdp.getHeight(); y++) {
                difference = stateUtilities[x][y] - oldStateUtilities[x][y];
                if (Math.abs(difference) > sigma) {
                    return false;
                }

            }
        }
        return true;
    }

    /*
    deep copy of utility array
     */
    private Double[][] duplicate(Double[][] stateUtilities) {
        Double[][] clone = new Double[stateUtilities.length][stateUtilities[0].
            length];
        for (int i = 0; i < stateUtilities.length; i++) {
            for (int j = 0; j < stateUtilities[0].length; j++) {
                clone[i][j] = stateUtilities[i][j];
            }
        }
        return clone;
    }

    public Action getPolicy(int x, int y) {
        return policy[x][y];
    }

    public double getExpectedValue(int x, int y){
        return stateUtilities[x][y] == null ? 0.0 : stateUtilities[x][y];
    }

    /**
     * the agent performs the moves following the computed policy.
     */
    public void ApplyPolicy() {
        mdp.setShowProgress(true);
        mdp.setWaittime(500);
        int xPos = 0;
        int yPos = 0;
        while (true) {
            xPos = mdp.getStateXPosition();
```

```java
            yPos = mdp.getStateYPostion();
            if (getPolicy(xPos, yPos) != null) {
                mdp.performAction(getPolicy(xPos, yPos));
            } else {
                break;
            }
        }
        System.out.println("finished");
    }

}
```

C Code for the Q-Learning algorithm

```java
package nl.ru.ai.vroon.mdp;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;

/**
 *
 * @author christianlammers
 */
public class QLearningAlgorithm {

    MarkovDecisionProblem mdp;
    double[][][] QValues;
    int counter = 0;
    private int maxIterations = 10000;
    private Random rnd;

    double discount = 0.9;
    double alpha = 0.2;

    double eps = 1.0;
    double eps_min = 0.01;
    double decay = 0.005;

    public QLearningAlgorithm(MarkovDecisionProblem m) {
        mdp = m;
        mdp.setInitialState(0, 0);
        rnd = new Random();
    }

    public void QLearning() {
        initializeQTable();

        for (int iter = 0; iter < maxIterations; iter++){

            int x = mdp.getStateXPosition();
            int y = mdp.getStateYPostion();
            while (mdp.getField(x, y) != Field.REWARD){
                try
                {Thread.sleep(0);}
                catch (Exception e)
                {e.printStackTrace();}

```

```
                int a = bestAction(x,y);
                double expFutValue = actionValue(a);
                double q = QValues[x][y][a];
                QValues[x][y][a] = q + alpha*(expFutValue - q);

                x = mdp.getStateXPosition();
                y = mdp.getStateYPostion();

                eps += -decay;
                if (eps < eps_min)
                    eps = eps_min;
            }
            mdp.restart();
        }

        printTable();

        mdp.setShowProgress(true);
        mdp.setWaittime(500);
        applyPolicy();



    }

    private void initializeQTable() {
        QValues = new double[mdp.getWidth()][mdp.getHeight()][4];
        for (int x = 0; x < mdp.getWidth(); x++) {
            for (int y = 0; y < mdp.getHeight(); y++) {
                for (int z = 0; z < QValues[0][0].length;z++)
                QValues[x][y][z] = 0.0;
            }
        }

    }



    private int bestAction(int x, int y) {
        float exploitation = rnd.nextFloat();

        int[] possibleActions = getPossibleActions(x,y);
        int k = 0;

        if (exploitation > eps){ //exploitation{

        //best action
        double bestValue = -999999999999.0;
        for (int a : possibleActions){
            if (QValues[x][y][a] > bestValue){
                bestValue = QValues[x][y][a];
                k = a;
            }
        }
        return k;
```

```java
    }
        else { //exploration
            k = rnd.nextInt(possibleActions.length);
            return possibleActions[k];
        }

        //random action
        //possibleActions[k];
    }

    private double actionValue(int a) {
        double reward = 0;
        switch(a){
            case 0: reward = mdp.performAction(Action.UP);
            break;
            case 1: reward = mdp.performAction(Action.RIGHT);
            break;
            case 2: reward = mdp.performAction(Action.DOWN);
            break;
            case 3: reward = mdp.performAction(Action.LEFT);
        }

        int newX = mdp.getStateXPosition();
        int newY = mdp.getStateYPostion();

        double maxAction = Arrays.stream(QValues[newX][newY]).max().getAsDouble(); //getBestFutureAction(newX, newY);
        //System.out.println("x: " + newX + " y: " + newY + " value: " + maxAction);

        return reward + discount*maxAction;
    }

    //public double getExpectedValue(int x, int y)

    private void printTable() {
        for (int i = 0; i < QValues.length; i++){
            for (int j =0; j < QValues[0].length; j++){
                for (int z = 0; z < QValues[0][0].length;z++){
                    System.out.println("Value for " + "x: " + i + " y: " + j + " action: " + z + " : " + QValues[i][j][z]);
                }
            }
        }
    }

    private int[] getPossibleActions(int x, int y) {
        ArrayList<Integer> possibleActions = new ArrayList<>();
        for (int a = 0; a < QValues[0][0].length; a++){
            if (isPossible(x,y,a))
                possibleActions.add(a);
        }
        return possibleActions.stream().mapToInt(i->i).toArray();
    }

    private boolean isPossible(int x,int y,int a) {
        switch(a){
            case 0: return y+1 < mdp.getHeight() && mdp.getField(x, y+1) != Field.OBSTACLE;
```

```
                case 1: return x+1 < mdp.getWidth() && mdp.getField(x+1, y) != Field.
                    OBSTACLE;
                case 2: return y-1 >= 0 && mdp.getField(x, y-1) != Field.OBSTACLE;
                case 3: return x-1 >= 0 && mdp.getField(x-1, y) != Field.OBSTACLE;
            }
            return true;
        }

        private double getBestFutureAction(int x, int y) {
            double[] actionValues = new double[4];
            actionValues[0] = QValues[x][y][0] * mdp.getpPerform() + QValues[x][y][1]
                * mdp.getPSideStep()/2 + QValues[x][y][2] * mdp.getpBackstep() +
                QValues[x][y][3] * mdp.getPSideStep()/2;
            actionValues[1] = QValues[x][y][1] * mdp.getpPerform() + QValues[x][y][0]
                * mdp.getPSideStep()/2 + QValues[x][y][2] * mdp.getPSideStep()/2 +
                QValues[x][y][3] * mdp.getpBackstep();
            actionValues[2] = QValues[x][y][2] * mdp.getpPerform() + QValues[x][y][1]
                * mdp.getPSideStep()/2 + QValues[x][y][3] * mdp.getPSideStep()/2 +
                QValues[x][y][0] * mdp.getpBackstep();
            actionValues[3] = QValues[x][y][3] * mdp.getpPerform() + QValues[x][y][0]
                * mdp.getPSideStep()/2 + QValues[x][y][2] * mdp.getPSideStep()/2 +
                QValues[x][y][1] * mdp.getpBackstep();

            return Arrays.stream(actionValues).max().getAsDouble();
        }

        private void applyPolicy() {
            int x = mdp.getStateXPosition();
            int y = mdp.getStateYPostion();
            while (mdp.getField(x, y) != Field.REWARD && mdp.getField(x, y) != Field.
                NEGREWARD){
                int bestAction = 0;
                double bestValue = -9999999;
                int[] possibleActions = getPossibleActions(x,y);
                for (int a : possibleActions){
                    if (QValues[x][y][a] > bestValue){
                        bestValue = QValues[x][y][a];
                        bestAction = a;
                    }
                }
                switch (bestAction){
                    case 0: mdp.performAction(Action.UP);
                    break;
                    case 1: mdp.performAction(Action.RIGHT);
                    break;
                    case 2: mdp.performAction(Action.DOWN);
                    break;
                    case 3: mdp.performAction(Action.LEFT);
                    break;
                }

                x = mdp.getStateXPosition();
                y = mdp.getStateYPostion();
            }
        }
}
```